EXHIBIT B

US005892914A

United States Patent [19]
Pitts

[11] **Patent Number: 5,892,914**
[45] **Date of Patent: *Apr. 6, 1999**

[54] **SYSTEM FOR ACCESSING DISTRIBUTED DATA CACHE AT EACH NETWORK NODE TO PASS REQUESTS AND DATA**

[76] Inventor: **William Michael Pitts**, 780 Mora Dr., Los Altos, Calif. 94024

[*] Notice: The term of this patent shall not extend beyond the expiration date of Pat. No. 5,611,049.

[21] Appl. No.: **806,441**

[22] Filed: **Feb. 26, 1997**

Related U.S. Application Data

[62] Division of Ser. No. 343,477, filed as PCT/US92/04939 Jun. 3, 1992, published as WO93/24890 Dec. 9, 1993, Pat. No. 5,611,049.

[51] **Int. Cl.**^{6} **G06F 15/163**
[52] **U.S. Cl.** **395/200.49**; 395/200.33; 395/876; 711/137
[58] **Field of Search** 711/137; 395/200.33, 395/200.48, 876, 200.49

[56] **References Cited**

U.S. PATENT DOCUMENTS

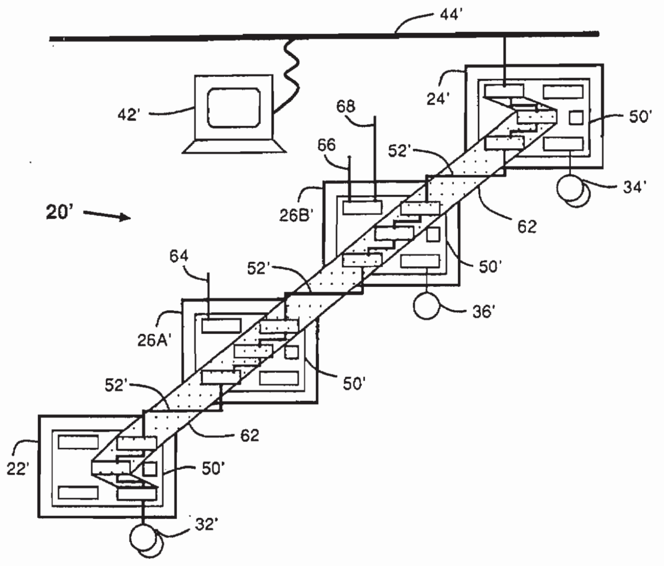
4,422,145	12/1983	Sacco et al.	395/650
4,463,424	7/1984	Mattson et al.	395/463
4,694,396	9/1987	Weisshaar et al.	395/200.03
4,714,992	12/1987	Gladney et al.	395/600
4,897,781	1/1990	Chang et al.	395/600
4,975,830	12/1990	Gerpheide et al.	395/200.1
5,001,628	3/1991	Johnson et al.	395/600
5,056,003	10/1991	Hammer et al.	395/650
5,058,000	10/1991	Cox et al.	395/600
5,077,658	12/1991	Bendert et al.	395/600
5,109,515	4/1992	Laggis et al.	395/650
5,113,519	5/1992	Johnson et al.	395/600
5,133,053	7/1992	Johnson et al.	395/200.03
5,155,808	10/1992	Shimizu	395/200.03
5,224,205	6/1993	Dinkin et al.	395/200.2

Primary Examiner—Kenneth S. Kim
Attorney, Agent, or Firm—Donald E. Schreiber

[57] **ABSTRACT**

Network Distributed Caches ("NDCs") (**50**) permit accessing a named dataset stored at an NDC server terminator site (**22**) in response to a request submitted to an NDC client terminator site (**24**) by a client workstation (**42**). In accessing the dataset, the NDCs (**50**) form a NDC data conduit (**62**) that provides an active virtual circuit ("AVC") from the NDC client site (**24**) through intermediate NDC sites (**26**B, **26**A) to the NDC server site (**22**). Through the AVC provided by the conduit (**62**), the NDC sites (**22**, **26**A and **26**B) project an image of the requested portion of the named dataset into the NDC client site (**24**) where it may be either read or written by the workstation **42**. The NDCs (**50**) maintain absolute consistency between the source dataset and its projections at all NDC client terminator sites (**24**, **204**B and **206**) at which client workstations access the dataset. Channels (**116**) in each NDC (**50**) accumulate profiling data from the requests to access the dataset for which they have been claimed. The NDCs (**50**) use the accumulated profile data stored in channels (**116**) to anticipate future requests to access datasets, and, whenever possible, prevent any delay to client workstations in accessing data by asynchronously pre-fetching the data in advance of receiving a request from a client workstation.

16 Claims, 22 Drawing Sheets



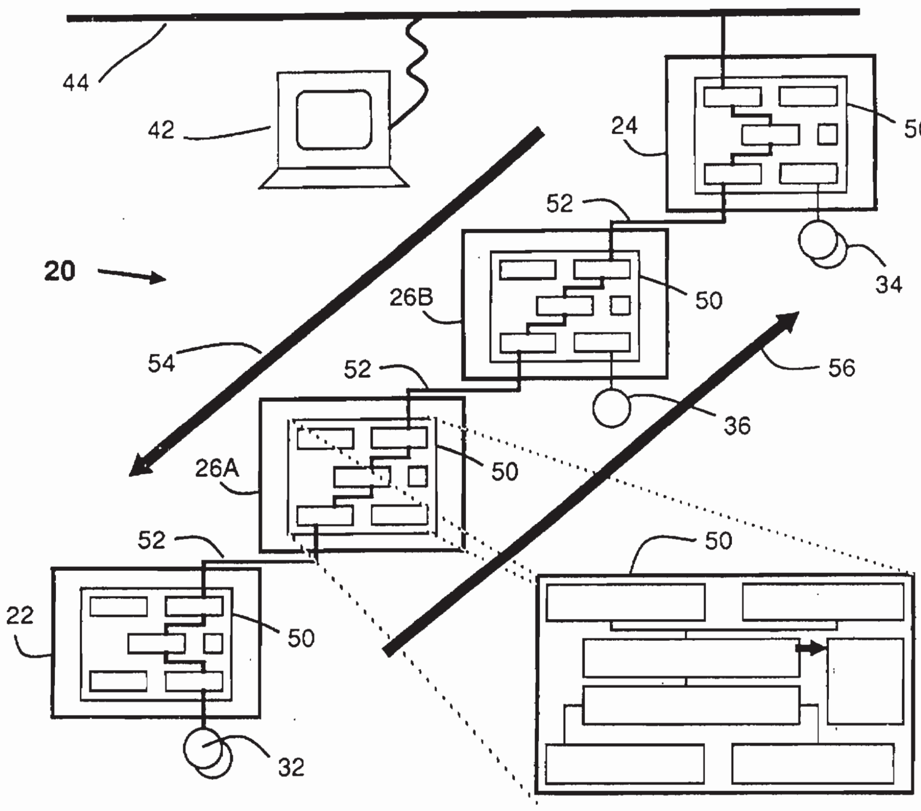


FIG. 1

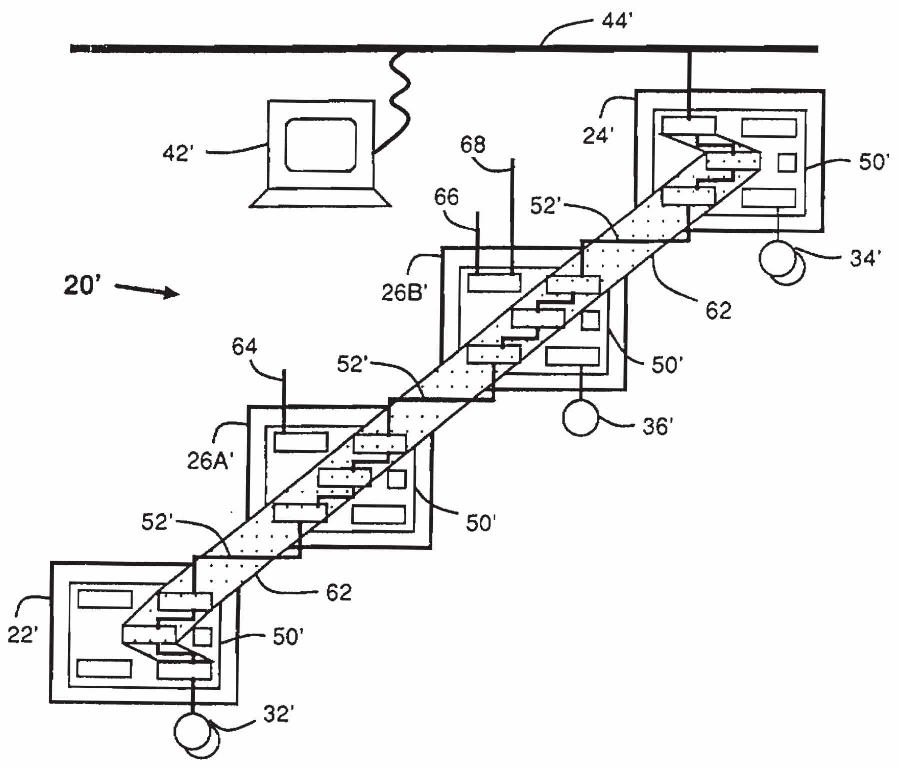


FIG. 2

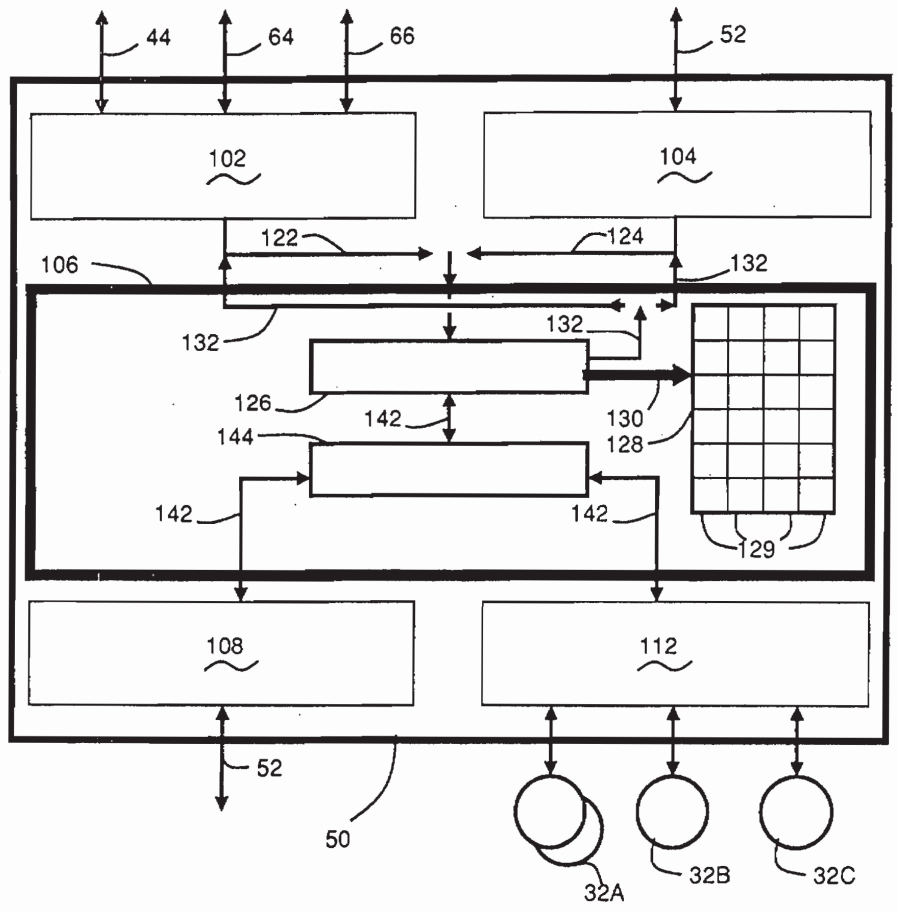

44
64
66
52
102
104
122
124
106
132
132
132
126
142
130
128
144
142
142
129
108
112
52
50
32A
32B
32C

FIG. 3

```
/************************************************************************
 * Channel Structure:
 ***********************************************************************/

typedef struct channel {
    u_long          c_flags;        /* see defines below                    */
    struct channel  *c_forw;        /* hash chain ptrs                      */
    struct channel  *c_back;
    struct channel  *av_forw;       /* free list ptrs                       */
    struct channel  *av_back;
    u_long          c_state;        /* see defines below                    */
    struct channel  *c_head;        /* pointer to primary channel           */
    NDC_FH          fh;             /* file handle                          */
    u_short         c_error;        /* channel error code                   */
    u_short         resid_err;      /* unreported error from prev op        */
    u_short         flush_level;    /* channel flush level                  */
    u_short         s_spare1;       /*                                      */
    u_long          refresh;        /* time of last attributes refresh      */
    u_long          c_size;         /* size of file, channel's view         */
    time_t          atime;          /* local time of last access            */
    time_t          mtime;          /* local time of last modification      */
    time_t          ctime;          /* local time file created              */
    NDC_STATS       stats;          /* filesystem stats                     */
    NDC_ATTR        attr;           /* cached copy of attributes            */
    NDC_USS         *uss;           /* ptr to list of upstream sites        */
    INODE           *ip;            /* inode pointer                        */
    struct channel  *server_cp;     /* downstream server's channel ptr      */
    NDC_PID         server_addr;    /* address of the downstream site       */
    NDC_MSG         *req_msg;       /* request msg being processed          */
    NDC_MSG         *msg_up;        /* upstream msg chain ptr               */
    NDC_MSG         *msg_down;      /* downstream msg chain ptr             */
    SUBCHANNEL      sc;             /* 1st subchannel (built in)            */
    RATE            c_rate;         /* channel data rate                    */
    RATE            f_rate;         /* file data rate                       */
    int             relse_time;     /* time: channel placed on free list    */
    int             splice_pnt;     /* channel splice point                 */
    int             splice_cnt;     /* channel splice count                 */
} CHANNEL;
```

FIG. 4A

```
typedef struct subchannel {
    struct channel     *next;          /* ptr to next subchannel              */
    struct channel     *ext;           /* ptr to subchannel extension         */
    long               offset;         /* offset to start of segment          */
    long               seg_length;     /* amount of data in this subchannel   */
    long               ext_length;     /* amount of data in this extent       */
    long               splice_pnt;     /* expected start of next request      */
    RATE               rate;           /* client data rate                    */
    long               bufcount;       /* # of buffers in bp[] array          */
    NDC_BD             bd[NDC_MAX_BDS_SC]; /* buffer descriptor array         */
} SUBCHANNEL;

typedef struct {
    long               flags;          /* flags: see below                    */
    u_short            leader;         /* # of invalid leading bytes          */
    u_short            trailer;        /* # of invalid trailing bytes         */
    vme_t              addr;           /* address of buffer                   */
    struct buf         *bp;            /* address of buffer header            */
} NDC_BD;


/*  NDC_BD.flags:      */

#define BD_RECEIVE_DATA    0x0001   /* buffer expecting WRITE data      */
#define BD_FRAG_REPLACE    0x0002   /* frag was REPLACEd                */
#define BD_FRAG_EXTEND     0x0004   /* frag was EXTENDed                */
#define BD_DIRTY_DATA      0x0008   /* buffer is DIRTY                  */

#define BD_INVAL           0x0010   /* buffer is INVALID                */
#define BD_MAPPED          0x0020   /* buffer has been MAPPED           */
#define BD_FLUSHED         0x0040   /* buffer is being FLUSHed          */
#define BD_CONJURED_BUF    0x0080   /* buffer was CONJURED              */
```

FIG. 4B

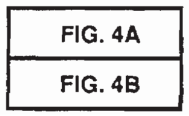


FIG. 4

```
/*
 * CHANNEL.c_flags:
 */
#define C_READ            0x00000001  /* reading                          */
#define C_WRITE           0x00000002  /* writing                          */
#define C_DECEASE         0x00000004  /* kill channel on release          */
#define C_P_FLUSH         0x00000008  /* partial flush has been received  */
#define C_EXCL            0x00000010  /* exclusive create                 */
#define C_ASYNC           0x00000020  /* client says "don't wait"         */
#define C_NOCACHE         0x00000040  /* discard channel on release       */
#define C_ASYNC_IO        0x00000080  /* don't wait for I/O completion    */
#define C_EOF             0x00000100  /* attempted to LOAD past the EOF   */
#define C_BUSY            0x00000200  /* not on av_forw/back list         */
#define C_ERROR           0x00000400  /* error occurred, cp->c_error set  */
#define C_BLOCKED         0x00000800  /* awaiting response to a request   */
#define C_LOAD_AGAIN      0x00001000  /* all data NOT delivered, try again */
#define C_RELOAD          0x00002000  /* second pass through ndc_load()   */
#define C_XXX6            0x00004000  /*                                  */
#define C_XXX7            0x00008000  /*                                  */

#define C_DIRTY_DATA      0x00010000  /* channel data has been modified   */
#define C_DIRTY_ATTRS     0x00020000  /* channel attrs have been modified */
#define C_SOILED_ATTRS    0x00040000  /* attr times have been modified    */
#define C_DELAYED_WRITE0x00080000     /* channel is DIRTY                 */
#define C_CACHE_DATA      0x00100000  /* requested data found in the cache */
#define C_CACHE_ATTRS     0x00200000  /* requested meta found in the cache */
#define C_DATA_VALID      0x00400000  /* valid data present               */
#define C_ATTRS_VALID     0x00800000  /* valid attrs present              */
#define C_WANTED          0x01000000  /* issue wakeup when BUSY goes off  */
#define C_LOCKED          0x02000000  /* locked in core (not LRUable)     */
#define C_EMPTY           0x04000000  /* channel has no buffers assigned  */
#define C_DESTROY         0x08000000  /* channel is marked for destruction */
#define C_XXX8            0x10000000  /*                                  */
#define C_SUBCHANEXT      0x20000000  /* this is a subchannel extension   */
#define C_SUBCHANNEL      0x40000000  /* this is a subchannel             */
#define C_HEAD            0x80000000  /* a channel header, not a channel  */


#define CHANNEL_DIRTY    ( C_DIRTY_DATA | C_DIRTY_ATTRS )

#define NDC_PROPAGATE_UP_FLAGS
#define NDC_PROPAGATE_DOWN_FLAGS    (C_EXCL|C_ASYNC|C_NOCACHE)
```

FIG. 5

```
/*
 * CHANNEL.c_state:
 */
#define NDC_SITE_READING                  C_READ
#define NDC_SITE_WRITING                  C_WRITE
#define NDC_SITE_ACTIVE                   (C_READ | C_WRITE)

#define NDC_SITE_IS_CCS                   0x00000010
#define NDC_SITE_CACHING_ENABLED          0x00000020

#define NDC_SITE_REQUEST_REJECTED         0x00000100
#define NDC_SITE_ACCEPT_REJECTEE          0x00000200
#define NDC_SITE_DEADLY_EMBRACE_REJECTED  0x00000400

#define NDC_SITE_CLIENT_TERM              0x00001000
#define NDC_SITE_SERVER_TERM              0x00002000

#define NDC_SITE_ALL_DEAD                 0x00010000
#define NDC_SITE_NEW_CHANNEL              0x00020000

#define NDC_SITE_NEEDS_SERVICE            0x00100000
#define NDC_SITE_CLIENT_WRITE_THRU        0x00200000

#define NDC_SITE_REBOOTING                0x20000000
#define NDC_SITE_OFFLINE                  0x40000000
#define NDC_SITE_ONLINE                   0x80000000

/*
 * USE_ONCE & USE_MANY are defined to keep in sync with
 * external documentation.
 */
#define USE_ONCE                          0x00000000
#define USE_MANY                      NDC_SITE_CACHING_ENABLED
```

FIG. 6

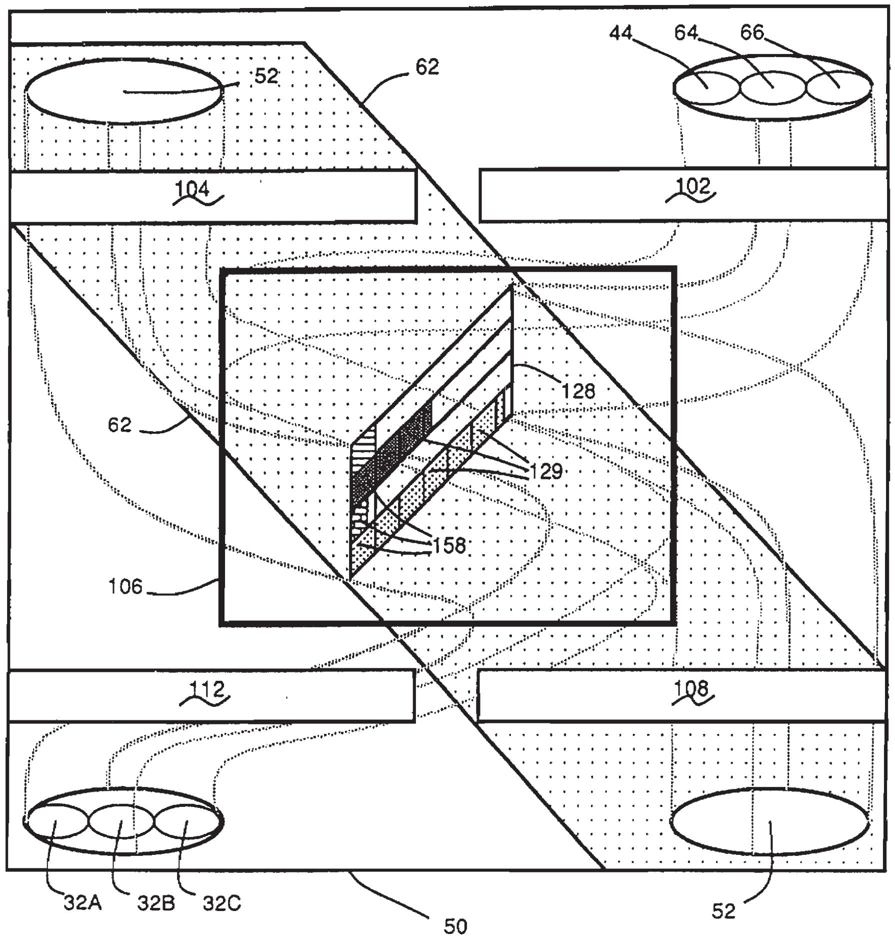


FIG. 7

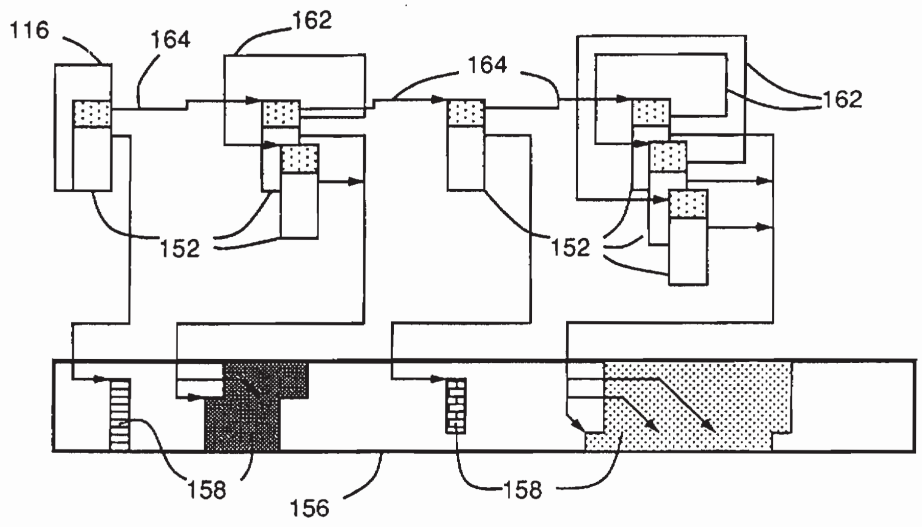


FIG. 8

```
/**************************************************************************
 * Data Transport Protocol:
*************************************************************************/

/*
 * NDC_MSG.type:
 */

#define NDC_ID                  ( (long)( ('W'<<8) | ('H') ) << 16 )

/* Configuration: */

#define NDC_GET_VERSION_NO  (  1 | NDC_ID )
#define NDC_MOUNT           (  2 | NDC_ID )
#define NDC_UNMOUNT         (  3 | NDC_ID )

/* Data Transfer: */

#define NDC_LOAD            (  4 | NDC_ID )
#define NDC_FLUSH           (  5 | NDC_ID )

#define NDC_LOAD_RELEASE    (  8 | NDC_ID )
#define NDC_FLUSH_RELEASE   (  9 | NDC_ID )

/* Control:       */

#define NDC_LOOKUP          ( 16 | NDC_ID )
#define NDC_CREATE          ( 17 | NDC_ID )
#define NDC_REMOVE          ( 18 | NDC_ID )
#define NDC_RENAME          ( 19 | NDC_ID )
#define NDC_LINK            ( 20 | NDC_ID )
#define NDC_SYMLINK         ( 21 | NDC_ID )
#define NDC_RMDIR           ( 22 | NDC_ID )
#define NDC_STATFS          ( 23 | NDC_ID )
#define NDC_FSYNC           ( 24 | NDC_ID )
#define NDC_ACCESS          ( 25 | NDC_ID )
#define NDC_SYNCFS          ( 26 | NDC_ID )
#define NDC_QUOTA           ( 27 | NDC_ID )

#define NDC_DISABLE         ( 30 | NDC_ID )
#define NDC_RECALL          ( 31 | NDC_ID )
```

FIG. 9

```
typedef struct ndc_msg {
    long        type;                          /* message type               */
    short       error;                         /* error type                 */
    u_short     flags;                         /* message flags              */
    union {
        union {                                /* NFS Proc #'s:              */
            NDC_MOUNT_INFO      marg;          /*                            */
            NDC_FH              fh;            /* 1, 5, 17                   */
            NDC_ATTR            sarg;          /* 2                          */
            NDC_DIROP_ARGS      darg;          /* 4, 10, 15                  */
            NDC_LOAD_ARGS       rdarg;         /* 6                          */
            NDC_FLUSH_ARGS      flarg;         /* 8                          */
            NDC_RELEASE_ARGS    relarg;        /*                            */
            NDC_CREATE_ARGS     carg;          /* 9, 14                      */
            NDC_RENAME_ARGS     rarg;          /* 11                         */
            NDC_LINK_ARGS       larg;          /* 12                         */
            NDC_SYMLINK_ARGS    slarg;         /* 13                         */
            NDC_PARTITION       part;          /*                            */
            NDC_DAEMON_TASK     task;          /* task for daemon            */
            NDC_MSG_CHAIN       mc;            /* message chain              */
            NDC_SP_RW           sp;            /* SP read/write              */
            NDC_CC_RECALL_ARGS  rcarg;         /* recall/disable caches      */
        }           in;
        union {                                /* NFS Proc #'s:              */
            NDC_MOUNT_INFO      mres;          /*                            */
            NDC_STATS           stats;         /* 17                         */
            NDC_ATTR            attr;          /* 1, 2                       */
            NDC_DIROP_RES       dirres;        /* 4, 9, 14                   */
            NDC_DATA            data;          /* 5                          */
            NDC_LOAD_RES        rdres;         /* 6                          */
            NDC_FLUSH_RES       flres;         /*                            */
            NDC_PARTITION       part;          /*                            */
            NDC_DAEMON_TASK     task;          /* task for daemon            */
            NDC_MSG_CHAIN       mc;            /* message chain              */
            NDC_SP_RW           sp;            /* SP read/write              */
            NDC_CC_RECALL_RES   rcres;         /* recall/disable caches      */
        }           out;
    }       un;
} NDC_MSG;

#define req     un.in
#define rsp     un.out
```

FIG. 10A

```
/*
 *  NDC_MSG.flags:
 */

/*
 *            consistency control
 */
#define NDC_MSG_SITE_READING        0x0001              /* down       */
#define NDC_MSG_SITE_WRITING        0x0002              /* down       */
#define NDC_MSG_SITE_RW_MASK        0x0003              /* down       */
#define NDC_MSG_SITE_RELEASE        0x0004              /* down       */
#define NDC_MSG_SITE_DECEASE        0x0008              /* down       */
#define NDC_MSG_SITE_ENABLED        0x0010              /* up         */
/*
 *            request modifiers
 */
#define NDC_REQ_EXCL                0x0020              /* down       */
#define NDC_REQ_ASYNC               0x0040              /* down       */
#define NDC_REQ_NOCACHE             0x0080              /* down       */
/*
 •            request specific flags (overlaid)
 */
#define NDC_MSG_XACT_DONE           0x0100              /* recall     */
#define NDC_LOAD_ATTRS_ONLY         0x0100              /* read       */
#define NDC_MSG_DATA_IS_COMMON      0x0200              /* read       */
#define NDC_MSG_CONVERT_DATA        0x0400              /* readdir    */
/*
 *            response status
 */
#define NDC_MSG_DATA_COMPLETE       0x1000              /* up/down    */
#define NDC_MSG_ATTRS_PRESENT       0x2000              /* up/down    */
#define NDC_RSP_REQUEST_REJECT      0x4000              /* up         */
#define NDC_MSG_DONE                0x8000              /* local      */
```

FIG. 10B

FIG. 10

```
typedef struct filehandle {
    long        fsid;               /* filesystem id                  */
    long        fid;                /* file id                        */
    long        gen;                /* generation number              */
} NDC_FH;

typedef long    NDC_PID;
```

FIG. 11A

```
/*
 * NDC_DATA.flags:
 */
#define NDC_DATA_DIRECT      0x00   /* data present within structure   */
#define NDC_DATA_INDIRECT    0x01   /* pointer to data is present       */

typedef struct {
    u_short     flags;
    u_short     len;
    union {
      vme_t       ptr;
      u_char      bytes;
    } un;
} NDC_DATA;                         /* size: 8 or (4 + len)             */
```

FIG. 11B

```
typedef struct {
    long        fsid;               /* File System ID                   */
    long        s_num;              /* Server mgr num. (i.e. 0, 1, ...) */
    NDC_PID     s_pid;              /* Server manager to own partition  */
    NDC_PID     sp_pid;             /* PID of SP that partition is on   */
    long        drive_set;          /* Which bank of drives is disk in  */
    long        disk_num;           /* Whic disk in bank is partition on */
    long        base;               /* Base of partition in 512 byte blocks */
    long        len;                /* Length of partition in 512 byte blks */
} NDC_PARTITION;
```

FIG. 11C

```
typedef struct {
    u_char      spare;           /* spare                               */
    u_char      status;          /* attributes status (see below)       */
    u_short     mode;            /* file's access mode and type         */
    uid_t       uid;             /* owner user id                       */
    gid_t       gid;             /* owner group id                      */
    u_long      size;            /* file size in bytes                  */
    time_t      atime;           /* time of last access                 */
    time_t      mtime;           /* time of last modification           */
} NDC_SATTR;                     /* size: 20                            */

/*
 * NDC_ATTR.status:
 */
#define F_ACC                    0x01  /* file has been accessed        */
#define F_UPD                    0x02  /* file has been modified        */
#define F_CHG                    0x04  /* inode has been changed        */
#define F_XXX                    0x08  /*                               */

#define F_ACC_SET                0x10  /* inode access time set by client */
#define F_UPD_SET                0x20  /* inode modify time set by client */
#define F_SIZE_SET               0x40  /* setattr changed the file size   */
#define F_LOCK_SET               0x80  /* set file lock                   */
```

FIG. 11D

```
typedef struct {
    NDC_PID        daemon_pid;   /* daemon assigned to this message     */
    struct ndc_msg *msg_ptr;     /* message pointer                     */
    struct channel  *cp;         /* channel pointer                     */
} NDC_DAEMON_TASK;               /* size: 12                            */
```

FIG. 11E

```
typedef struct ndc_upstream_site {
    struct ndc_up_site *next;    /* pointer to next uss                 */
    short       error;           /* error, if any, upstream             */
    short       spare;
    short       current_state;   /* what we think's happening           */
    short       actual_state;    /* what's really happening             */
    NDC_PID     upstream_pid;    /* address of upstream site            */
} NDC_USS;
```

FIG. 11F

```
/*
 * Buffer Descriptor:
 *
 * An NDC_BUF_DESC uses the high order bits to address the
 * common memory data buffer.  Since the buffers are aligned on
 * block boundaries (8K and likely to grow in the future), the low
 * order bits can be used to specify:
 *
 *  o  the segment number to which the buffer belongs for READs,
 *
 *  o  the number of invalid trailing bytes in the buffer for WRITEs.
 *
 *       NDC_BUF_DESC:   AAAAAAAA AAAAAAAA AAASSSSS SSSSSSSS
 */
typedef long                        NDC_BUF_DESC;


/*
 * Segment Descriptor:
 */
typedef struct {
    long          offset;
    long          count;
} NDC_SD;                           /* size: 8                               */


/*
 * Flush Descriptor:
 */
typedef struct {
    long                 offset;    /* offset, need not be block aligned     */
    NDC_BUF_DESC  bd;               /* buffer descriptor                     */
} NDC_FLUSH_DESC;                   /* size: 8                               */
```

FIG. 11G

```
typedef struct {
    struct channel      *cp;          /* channel ptr: read_more, read_relse      */
    NDC_FH              fh;           /* file handle                             */
    long                no_segs;      /* # of segments to read                   */
    NDC_SD              sd[MAX_SEGS]; /*  segment descriptors                    */
} NDC_LOAD_ARGS;                      /* size: 20 + (8 * X)                      */

typedef struct {
    struct channel      *cp;          /* channel ptr: read_more, read_relse      */
    short               bd_cnt;       /* # of bd[]s being returned               */
    short               seg_cnt;      /* # of segments being returned            */
    NDC_BUF_DESC        bd[MAX_BDS];  /* buf descriptors                         */
    NDC_ATTR            attr;         /* note: bd[] may overrun attrs when       */
                                      /*  NDC_RSP_ATTRS_PRESENT is not set       */
} NDC_LOAD_RES;                       /* size: 56 + (4 * X)                      */

typedef struct {
    struct channel      *cp;          /* chan ptr: flush_more, flush_relse       */
    NDC_FH              fh;           /* file handle                             */
    short               level;        /* flush level                             */
    short               no_fds;       /* # of fd[]s being flushed                */
    NDC_SATTR           sattr;        /* file attributes (writable)              */
    NDC_FLUSH_DESC      fd[MAX_FDS];  /* flush descriptors                       */
} NDC_FLUSH_ARGS;                     /* size: 40 + (8 * X)                      */

typedef struct {
    struct channel      *cp;          /* chan ptr: flush_more, flush_relse       */
    NDC_FH              fh;           /* file handle                             */
    short               level;        /* flush level                             */
    short               no_fds;       /* # of fd[]s being flushed                */
    NDC_ATTR            attr;         /* file attributes                         */
} NDC_FLUSH_RES;                      /* size: 68                                */

typedef struct {
    struct channel      *cp;          /* channel pointer                         */
    NDC_FH              fh;           /* file handle                             */
    short               error;        /* error                                   */
} NDC_RELEASE_ARGS;                   /* size: 18                                */
```

FIG. 11H

```
typedef struct {
    NDC_FH              fh;
    NDC_DATA            name;
} NDC_DIROP_ARGS;                   /* size: 20 or 16 + NDC_DATA.len        */

typedef struct {
    NDC_FH              fh;
    NDC_ATTR            attr;
} NDC_DIROP_RES;                    /* size: 60                             */

typedef struct {
    NDC_DIROP_ARGS      where;
    NDC_ATTR            attr;
} NDC_CREATE_ARGS;                  /* size: 68 or 64 + NDC_DATA.len        */

typedef struct {
    NDC_DIROP_ARGS      from;
    NDC_DIROP_ARGS      to;
} NDC_RENAME_ARGS;                  /* size: 2*(20 or 16 + NDC_DATA.len)    */

typedef struct {
    NDC_FH              from;
    NDC_DIROP_ARGS      to;
} NDC_LINK_ARGS;                    /* size: 32 or 28 + NDC_DATA.len        */

typedef struct {
    NDC_DIROP_ARGS      from;
    NDC_ATTR            attr;
    NDC_DATA            link;
} NDC_SYMLINK_ARGS;                 /* size: 76 or 68 + (2*NDC_DATA.len)    */

typedef struct {
    NDC_FH              file;       /* OUT: filesystem root file handle     */
    NDC_PARTITION       part;       /* IN/OUT: Describes partition to use   */
    NDC_DATA            path;       /* IN: mount point pathname             */
} NDC_MOUNT_INFO;                   /* size: 52 or 48 + NDC_DATA.len        */
```

FIG. 11I

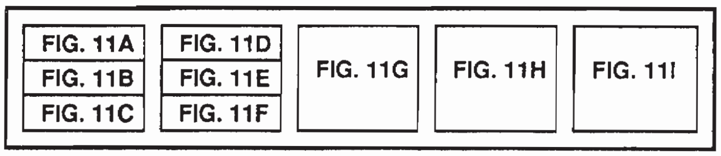


FIG. 11

```
/*
 * Message chains are used to group together a series of related messages
 * that are to be submitted and processed as an atomic unit.  A message chain
 * is "done" only when all messages have been dispatched to their respective
 * destinations and the response for each message has been received.  All
 * message types capable of being chained must align with the structure
 * NDC_MSG_CHAIN defined below.  The message types that employ chaining
 * are:  NDC_SP_RW, and NDC_CC_RECALL_ARGS/NDC_CC_RECALL_RES.
 */
typedef struct {
    struct channel      *msg_head_cp;
    struct ndc_msg      *next_msg;
} NDC_MSG_CHAIN;

typedef struct {
    struct channel      *msg_head_cp;
    struct ndc_msg      *next_msg;
    u_char              scsi_id;
    u_char              disk_number;
    short               sector_cnt;
    short               block_size;
    u_long              sector_adr;
    vme_t               vme_adr[NDC_SP_MAX_CONTIG];
} NDC_SP_RW;                            /* size: 20 + (4 * X)              */

typedef struct {
    struct channel      *msg_head_cp;
    struct ndc_msg      *next_msg;
    struct channel      *uss_cp;
    NDC_FH              fh;
    short               expected_state;
} NDC_CC_RECALL_ARGS;                   /* size: 26                        */

typedef struct {
    struct channel      *msg_head_cp;
    struct ndc_msg      *next_msg;
    struct channel      *uss_cp;
    short               actual_state;
    short               no_fds;         /* # of fd[]s being flushed        */
    long                size;           /* file size                       */
    time_t              mtime;          /* modification time               */
    NDC_FLUSH_DESC      fd[MAX_FDS];
} NDC_CC_RECALL_RES;                    /* size: 24 + (8 * X)              */
```

FIG. 12

```
typedef struct {
    u_char          vfs_flag;   /* filesystem flags                     */
    u_char          status;     /* attributes status (see below)        */
    u_short         b_limit;    /* block limit: !0 => max delta more    */
    u_short         type;       /* vnode type (for create)              */
    u_short         mode;       /* file's access mode and type          */
    uid_t           uid;        /* owner user id                        */
    gid_t           gid;        /* owner group id                       */
    short           nlink;      /* number of references to file         */
    dev_t           rdev;       /* device the file represents           */
    long            fsid;       /* file system id (dev for now)         */
    long            nodeid;     /* node id                              */
    u_long          size;       /* file size in bytes                   */
    long            blocks;     /* kbytes of disk space held by file    */
    long            blocksize;  /* blocksize preferred for i/o          */
    time_t          atime;      /* time of last access                  */
    time_t          mtime;      /* time of last modification            */
    time_t          ctime;      /* time file created                    */
} NDC_ATTR;                     /* size: 48                             */
```

FIG. 13A

```
typedef struct {
    long            bsize;      /* fundamental file system block size   */
    long            blocks;     /* total blocks in file system          */
    long            bfree;      /* free blocks in fs                    */
    long            bavail;     /* free blocks avail to non-superuser   */
    u_long          files;      /* total number of file slots           */
    u_long          ffree;      /* number of free file slots            */
} NDC_STATS;                    /* size: 24                             */
```

FIG. 13B

FIG. 13A
FIG. 13B

FIG. 13

```
typedef struct ndc_up_site {
    struct ndc_up_site *next;       /* pointer to next uss             */
    short        error;             /* error, if any, upstream         */
    short        spare;
    short        current_state;     /* what we think's happening       */  182
    short        actual_state;      /* what's really happening         */
    NDC_PID      upstream_pid;      /* addr of upstream site           */
} NDC_USS;                          /* size: 16                        */
```

FIG. 14

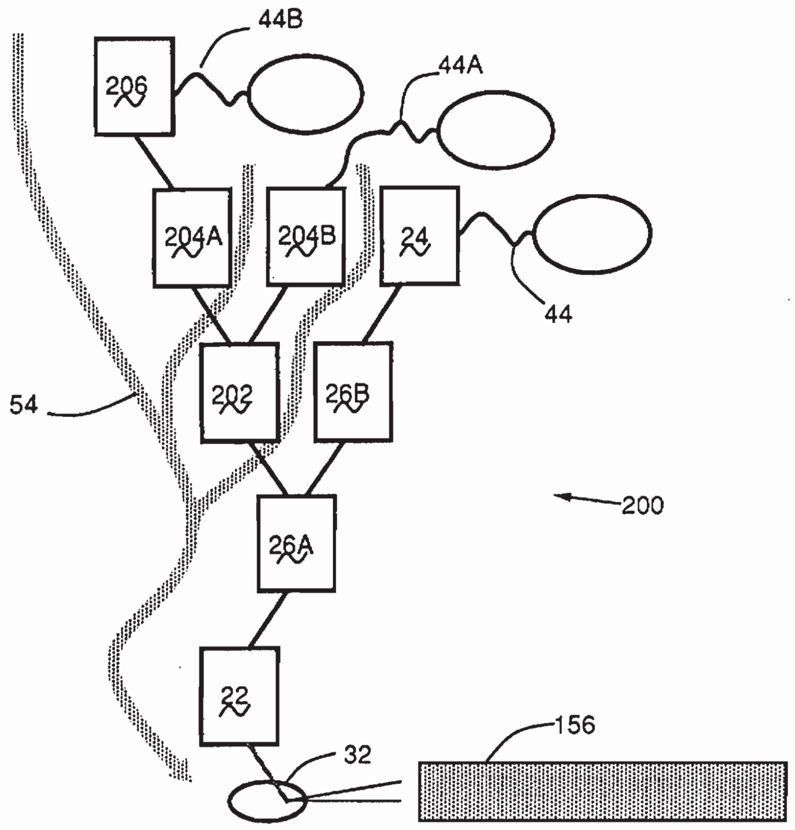


FIG. 15

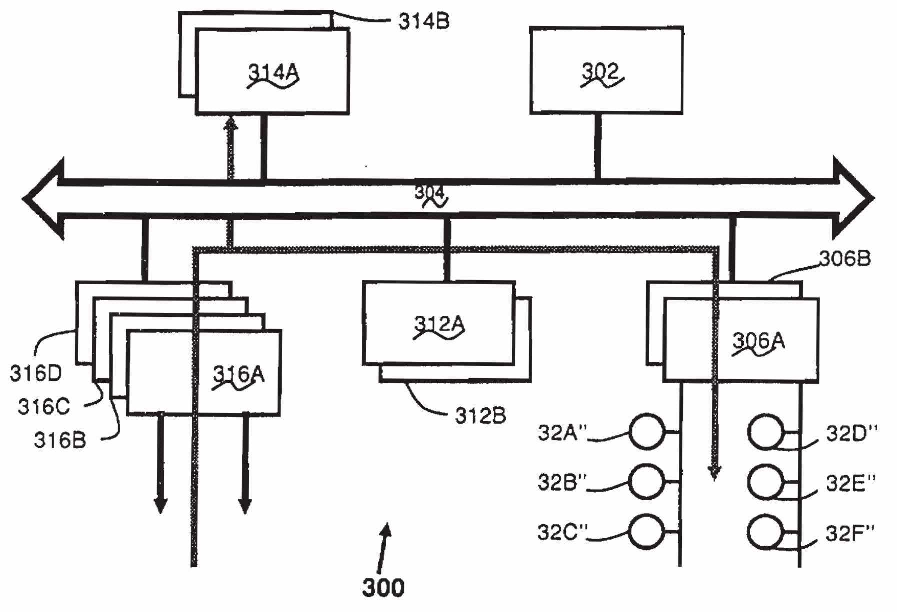

314B
314A
302
304
306B
316D
316C
316B
316A
312A
312B
306A
32A"
32B"
32C"
32D"
32E"
32F"
300

FIG. 16

SYSTEM FOR ACCESSING DISTRIBUTED DATA CACHE AT EACH NETWORK NODE TO PASS REQUESTS AND DATA

This is a division of application Ser. No. 08/343,477 filed Nov. 28, 1994, that issued Mar. 11, 1997, as U.S. Pat. No. 5,611,049, and that claimed priority under 35 U.S.C. § 371 from Patent Cooperation Treaty ("PCT") International Patent Application PCT/US92/04939 filed Jun. 3, 1992.

TECHNICAL FIELD

The present invention relates generally to the technical field of multi-processor digital computer systems and, more particularly, to multi-processor computer systems in which:

1. the processors are loosely coupled or networked together;
2. data needed by some of the processors is controlled by a different processor that manages the storage of and access to the data;
3. processors needing access to data request such access from the processor that controls the data;
4. the processor controlling data provides requesting processors with access to it.

BACKGROUND ART

Within a digital computer system, processing data stored in a memory; e.g., a Random Access Memory ("RAM") or on a storage device such as a floppy disk drive, a hard disk drive, a tape drive, etc.; requires copying the data from one location to another prior to processing: Thus, for example, prior to processing data stored in a file in a comparatively slow speed storage device such as hard disk, the data is first copied from the computer system's hard disk to its much higher speed RAM. After data has been copied from the hard disk to the RAM, the data is again copied from the RAM to the computer system's processing unit where it is actually processed. Each of these copies of the data, i.e., the copy of the data stored in the RAM and the copy of the data processed by the processing unit, can be considered to be image of the data stored on the hard disk. Each of these images of the data may be referred to as a projection of the data stored on the hard disk.

In a loosely coupled or networked computer system having several processors that operate autonomously, the data needed by one processor may be accessed only by communications passing through one or more of the other processors in the system. For example, in a Local Area Network ("LAN") such as Ethernet one of the processors may be dedicated to operating as a file server that receives data from other processors via the network for storage on its hard disk, and supplies data from its hard disk to the other processors via the network. In such networked computer systems, data may pass through several processors in being transmitted from its source at one processor to the processor requesting it.

In some networked computer systems, images of data are transmitted directly from their source to a requesting processor. One operating characteristic of networked computer systems of this type is that, as the number of requests for access to data increase and/or the amount of data being transmitted in processing each request increases, ultimately the processor controlling access to the data or the data transmission network becomes incapable of responding to requests within an acceptable time interval. Thus, in such networked computer systems, an increasing workload on the processor controlling access to data or on the data transmission network ultimately causes unacceptably long delays between a processor's request to access data and completion of the requested access.

In an attempt to reduce delays in providing access to data in networked computer systems, there presently exist systems that project an image of data from its source into an intermediate storage location in which the data is more accessible than at the source of the data. The intermediate storage location in such systems is frequently referred to as a "cache," and systems that project images of data into a cache are be referred to as "caching" systems.

An important characteristic of caching systems, frequently referred to as "cache consistency" or "cache coherency," is their ability to simultaneously provide all processors in the networked computer system with identical copies of the data. If several processors concurrently request access to the same data, one processor may be updating the data while another processor is in the process of referring to the data being updated. For example, in commercial transactions occurring on a networked computer system one processor may be accessing data to determine if a customer has exceeded their credit limit while another processor is simultaneously posting a charge against that customer's account. If a caching system lacks cache consistency, it is possible that one processor's access to data to determine if the customer has exceeded their credit limit will use a projected image of the customer's data that has not been updated with the most recent charge. Conversely, in a caching system that possesses complete or absolute cache consistency, the processor that is checking the credit limit is guaranteed that the data it receives incorporates the most recent modifications.

One presently known system that employs data caching is the Berkeley Software Distribution ("BSD") 4.3 version of the Unix timesharing operating system. The BSD 4.3 system includes a buffer cache located in the host computer's RAM for storing projected images of blocks of data, typically 8 k bytes, from files stored on a hard disk drive. Before a particular item of data may be accessed on a BSD 4.3 system, the requested data must be projected from the hard disk into the buffer cache. However, before the data may be projected from the disk into the buffer cache, space must first be found in the cache to store the projected image. Thus, for data that is not already present in a BSD 4.3 system's buffer cache, the system must perform the following steps in providing access to the data:

Locate the buffer in the RAM that contains the Least Recently Used ("LRU") block of disk data.

Discard the LRU block of data which may entail writing that block of data back to the hard disk.

Project an image of the requested block of data into the now empty buffer.

Provide the requesting processor with access to the data.

If the data being accessed by a processor is already present in a BSD 4.3 system's data cache, then responding to a processor's request for access to data requires only the last operation listed above. Because accessing data stored in RAM is much faster that accessing data stored on a hard disk, a BSD 4.3 system responds to requests for access to data that is present in its buffer cache in approximately 1/250th the time that it takes to respond to a request for access to data that is not already present in the buffer cache.

The consistency of data images projected into the buffer cache in a BSD 4.3 system is excellent. Since the only path from processors requesting access to data on the hard disk is through the BSD 4.3 system's buffer cache, out of date blocks of data in the buffer cache are always overwritten by their more current counterpart when that block's data returns from the accessing processor. Thus, in the BSD 4.3 system an image of data in the system's buffer cache always reflects the true state of the file. When multiple requests contend for the same image, the BSD 4.3 system queues the requests from the various processors and sequences the requests such that each request is completely serviced before any processing commences on the next request. Employing the preceding strategy, the BSD 4.3 system ensures the integrity of data at the level of individual requests for access to segments of file data stored on a hard disk.

Because the BSD 4.3 system provides access to data from its buffer cache, blocks of data on the hard disk frequently do not reflect the true state of the data. That is, in the BSD 4.3 system, frequently the true state of a file exists in the projected image in the system's buffer cache that has been modified since being projected there from the hard disk, and that has not yet been written back to the hard disk. In the BSD 4.3 system, images of data that are more current than and differ from their source on the hard disk data may persist for very long periods of time, finally being written back to the hard disk just before the image is about to be discarded due to its "death" by the LRU process. Conversely, other caching systems exist that maintain data stored on the hard disk current with its image projected into a data cache. Network File System ("NFS®") is one such caching system.

In many ways, NFS's client cache resembles the BSD 4.3 systems buffer cache. In NFS, each client processor that is connected to a network may include its own cache for storing blocks of data. Furthermore, similar to BSD 4.3, NFS uses the LRU algorithm for selecting the location in the client's cache that receives data from an NFS server across the network, such as Ethernet. However, perhaps one of NFS's most significant differences is that images of blocks of data are not retrieved into NFS's client cache from a hard disk attached directly to the processor as in the BSD 4.3 system. Rather, in NFS images of blocks of data come to NFS's client cache from a file server connected to a network such as Ethernet.

The NFS client cache services requests from a computer program executed by the client processor using the same general procedures described above for the BSD 4.3 system's buffer cache. If the requested data is already projected into the NFS client cache, it will be accessed almost instantaneously. If requested data is not currently projected into NFS's client cache, the LRU algorithm must be used to determine the block of data to be replaced, and that block of data must be discarded before the requested data can be projected over the network from the file server into the recently vacated buffer.

In the NFS system, accessing data that is not present in its client cache takes approximately 500 times longer than accessing data that is present there. About one-half of this delay is due to the processing required for transmitting the data over the network from an NFS file server to the NFS client cache. The remainder of the delay is the time required by the file server to access the data on its hard disk and to transfer the data from the hard disk into the file server's RAM.

In an attempt to reduce this delay, client processors read ahead to increase the probability that needed data will have already been projected over the network from the file server into the NFS client cache. When NFS detects that a client processor is accessing a file sequentially, blocks of data are asynchronously pre-fetched in an attempt to have them

present in the NFS client cache when the client processor requests access to the data. Furthermore, NFS employs an asynchronous write behind mechanism to transmit all modified data images present in the client cache back to the file server without delaying the client processor's access to data in the NFS client cache until NFS receives confirmation from the file server that it has successfully received the data. Both the read ahead and the write behind mechanisms described above contribute significantly to NFS's reasonably good performance. Also contributing to NFS's good performance is its use of a cache for directories of files present on the file server, and a cache for attributes of files present on the file server.

Several features of NFS reduce the consistency of its projected images of data. For example, images of file data present in client caches are re-validated every 3 seconds. If an image of a block of data about to be accessed by a client is more than 3 seconds old, NFS contacts the file server to determine if the file has been modified since the file server originally projected the image of this block of data. If the file has been modified since the image was originally projected, the image of this block in the NFS client cache and all other projected images of blocks of data from the same file are removed from the client cache. When this occurs, the buffers in RAM thus freed are queued at the beginning of a list of buffers (the LRU list) that are available for storing the next data projected from the file server. The images of blocks of data discarded after a file modification are re-projected into NFS's client cache only if the client processor subsequently accesses them.

If a client processor modifies a block of image data present in the NFS client cache, to update the file on the file server NFS asynchronously transmits the modified data image back to the server. Only when another client processor subsequently attempts to access a block of data in that file will its cache detect that the file has been modified.

Thus, NFS provides client processors with data images of poor consistency at reasonably good performance. However, NFS works only for those network applications in which client processors don't share data or, if they do share data, they do so under the control of a file locking mechanism that is external to NFS. There are many classes of computer application programs that execute quite well if they access files directly using the Unix File System that cannot use NFS because of the degraded images projected by NFS.

Another limitation imposed by NFS is the relatively small size (8 k bytes) of data that can be transferred in a single request. Because of this small transfer size, processes executing on a client processor must continually request additional data as they process a file. The client cache, which typically occupies only a few megabytes of RAM in each client processor, at best, reduces this workload to some degree. However, the NFS client cache cannot mask NFS's fundamental character that employs constant, frequent communication between a file server and all of the client processors connected to the network. This need for frequent server/client communication severely limits the scalability of an NFS network, i.e., severely limits the number of processors that may be networked together in a single system.

Andrew File System ("AFS") is a data caching system that has been specifically designed to provide very good scalability. Now used at many universities, AFS has demonstrated that a few file servers can support thousands of client workstations distributed over a very large geographic area. The major characteristics of AFS that permit its scalability are:

The unit of cached data increases from NFS's 8 k disk block to an entire file. AFS projects complete files from the file server into the client workstations.

Local hard disk drives, required on all AFS client workstations, hold projected file images. Since AFS projects images of complete files, its RAM is quickly occupied by image projections. Therefore, AFS projects complete files onto a client's local hard disk, where they can be locally accessed many times without requiring any more accesses to the network or to the file server.

In addition to projecting file images onto a workstation's hard disk, similar to BSD 4.3, AFS also employs a buffer cache located in the workstation's RAM to store images of blocks of data projected from the file image stored on the workstation's hard disk.

Under AFS, when a program executing on the workstation opens a file, a new file image is projected into the workstation from the file server only if the file is not already present on the workstation's hard disk, or if the file on the file server supersedes the image stored on the workstation's hard disk. Thus, assuming that an image of a file has previously been projected from a network's file server into a workstation, a computer program's request to open that file requires, at a minimum, that the workstation transmit at least one message back to the server to confirm that the image currently stored on its hard disk is the most recent version. This re-validation of a projected image requires a minimum of 25 milliseconds for files that haven't been superseded. If the image of a file stored on the workstation's hard disk has been superseded, then it must be re-projected from the file server into the workstation, a process that may require several seconds. After the file image has been re-validated or re-projected, programs executed by the workstation access it via AFS's local file system and its buffer cache with response comparable to those described above for BSD 4.3.

The consistency of file images projected by AFS start out as being "excellent" for a brief moment, and then steadily degrades over time. File images are always current immediately after the image has been projected from the file server into the client processor, or re-validated by the file server. However, several clients may receive the same file projection at roughly the same time, and then each client may independently begin modifying the file. Each client remains completely unaware of any modifications being made to the file by other clients. As the computer program executed by each client processor closes the file, if the file has been modified the image stored on the processor's hard disk is transmitted back to the server. Each successive transmission from a client back to the file server overwrites the immediately preceding transmission. The version of the file transmitted from the final client processor to the file server is the version that the server will subsequently transmit to client workstations when they attempt to open the file. Thus at the conclusion of such a process the file stored on the file server incorporates only those modifications made by the final workstation to transmit the file, and all modifications made at the other workstations have been lost. While the AFS file server can detect when one workstation's modifications to a file overwrites modifications made to the file by another workstation, there is little the server can do at this point to prevent this loss of data integrity.

AFS, like NFS, fails to project images with absolute consistency. If computer programs don't employ a file locking mechanism external to AFS, the system can support only applications that don't write to shared files. This

characteristic of AFS precludes using it for any application that demands high integrity for data written to shared files.

DISCLOSURE OF INVENTION

An object of the present invention is to provide a digital computer system capable of projecting larger data images, over greater distances, at higher bandwidths, and with much better consistency than the existing data caching mechanisms.

Another object of the present invention is to provide a generalized data caching mechanism capable of projecting multiple images of a data structure from its source into sites that are widely distributed across a network.

Another object of the invention is to provide a generalized data caching mechanism in which an image of data always reflects the current state of the source data structure, even when it is being modified concurrently at several remote sites.

Another object of the present invention is to provide a generalized data caching mechanism in which a client process may operate directly upon a projected image as though the image were actually the source data structure.

Another object of the present invention is to provide a generalized data caching mechanism that extends the domain over which data can be transparently shared.

Another object of the present invention is to provide a generalized data caching mechanism that reduces delays in responding to requests for access to data by projecting images of data that may be directly processed by a client site into sites that are "closer" to the requesting client site.

Another object of the present invention is to provide a generalized data caching mechanism that transports data from its source into the projection site(s) efficiently.

Another object of the present invention is to provide a generalized data caching mechanism that anticipates future requests from clients and, when appropriate, projects data toward the client in anticipation of the client's request to access data.

Another object of the present invention is to provide a generalized data caching mechanism that maintains the projected image over an extended period of time so that requests by a client can be repeatedly serviced from the initial projection of data.

Another object of the present invention is to provide a generalized data caching mechanism that employs an efficient consistency mechanism to guarantee absolute consistency between a source of data and all projected images of the data.

Briefly the present invention in its preferred embodiment includes a plurality of digital computers operating as a network. Some of the computers in the network function as Network Distributed Cache ("NDC") sites. Operating in the digital computer at each NDC site is an NDC that includes NDC buffers. The network of digital computers also includes one or more client sites, which may or may not be NDC sites. Each client site presents requests to an NDC to access data that is stored at an NDC site located somewhere within the network. Each item of data that may be requested by the client sites belongs to a named set of data called a dataset. The NDC site storing a particular dataset is called the NDC server terminator site for that particular dataset. The NDC site that receives requests to access data from the client site is called the NDC client terminator site. A single client site may concurrently request to access different datasets that are respectively stored at different NDC sites.

Thus, while there is only a single NDC client terminator site for each client site, simultaneously there may be a plurality of NDC server terminator sites responding to requests from a single client site to access datasets stored at different NDC server terminator sites.

Each NDC in the network of digital computers receives requests to access the data in the named datasets. If this NDC site is an NDC client terminator site for a particular client site, it will receive requests from that client. However, the same NDC site that is an NDC client terminator site for one client, may also receive requests to access data from other NDC sites that may or may not be NDC client terminator sites for other client sites.

An NDC client terminator site, upon receiving the first request to access a particular named dataset from a client site, assigns a data structure called a channel to the request and stores information about the request into the channel. Each channel functions as a conduit through the NDC site for projecting images of data to sites requesting access to the dataset, or, if this NDC site is an NDC client terminator site for a particular request, the channel may store an image of the data in the NDC buffers at this NDC site. In addition to functioning as part of a conduit for transmitting data between an NDC server terminator site and an NDC client terminator site, each channel also stores data that provides a history of access patterns for each client site as well as performance measurements both for client sites and for the NDC server terminator site.

When an NDC site receives a request to access data, regardless of whether the request is from a client site or from another NDC site, the NDC first checks the NDC buffers at this NDC site to determine if a projected image of the requested data is already present in the NDC buffers. If the NDC buffers at this NDC site do not contain a projected image of all data requested from the dataset, and if the NDC site receiving the request is not the NDC server terminator site for the dataset, the NDC of this NDC site transmits a single request for all of the requested data that is not present at this NDC site from this NDC site downstream to another NDC site closer to the NDC server terminator site for the dataset than the present NDC site. If the NDC buffers of this NDC site do not contain a projected image of all data requested from the dataset, and if the NDC site receiving the request is the sever terminator site for the dataset, the NDC of this NDC site accesses the stored dataset to project an image of the requested data into its NDC buffers. The process of checking the NDC buffers to determine if a projected image of the requested data is present there, and if one is not completely present, requesting the additional required data from a downstream NDC site or accessing the stored dataset repeats until the NDC buffers of the site receiving the request contains a projected image of all requested data.

The process of one NDC site requesting data from another downstream NDC site establishes a chain of channels respectively located in each of the NDC sites that provides a conduit for returning the requested data back to the NDC client terminator site. Thus, each successive NDC site in this chain of NDC sites, having obtained a projected image of all the requested data, either by accessing the stored dataset or from its downstream NDC site, returns the data requested from it upstream to the NDC site from which it received the request. This sequence of data returns from one NDC site to its upstream NDC site continues until the requested data arrives at the NDC client terminator site. When the requested data reaches the NDC client terminator site for this request, that NDC site returns the requested data to the client site.

Thus, the network of digital computers, through the NDCs operating in each of the NDC sites in the network, may project images of a stored dataset from an NDC server terminator site to a plurality of client sites in response to requests to access such dataset transmitted from the client sites to NDC client terminator sites. Furthermore, each NDC includes routines called channel daemons that operate in the background in each NDC site. The channel daemons use historical data about accesses to the datasets, that the NDCs store in the channels, to pre-fetch data from the NDC server terminator site to the NDC client terminator site in an attempt to minimize any delay between the receipt of a request to access data from the client site and the response to that request by the NDC client terminator site.

In addition to projecting images of a stored dataset, the NDCs detect a condition for a dataset, called a concurrent write sharing ("CWS") condition, whenever two or more client sites concurrently access a dataset, and one or more of the client sites attempts to write the dataset. If a CWS condition occurs, one of the NDC sites declares itself to be a consistency control site ("CCS") for the dataset, and imposes restrictions on the operation of other NDC sites upstream from the CCS. The operating restrictions that the CCS imposes upon the upstream NDC sites guarantee client sites throughout the network of digital computers the same level of file consistency the client sites would have if all the client sites operated on the same computer. That is, the operating conditions that the CCS imposes ensure that modifications made to a dataset by one client site are reflected in the subsequent images of that dataset projected to other client sites no matter how far the client site modifying the dataset is from the client site that subsequently requests to access the dataset.

These and other features, objects and advantages will be understood or apparent to those of ordinary skill in the art from the following detailed description of the preferred embodiment as illustrated in the various drawing figures.

BRIEF DESCRIPTION OF DRAWINGS

FIG. **1** is a block diagram depicting a networked, multiprocessor digital computer system that includes an NDC server terminator site, an NDC client terminator site, and a plurality of intermediate NDC sites, each NDC site in the networked computer system operating to permit the NDC client terminator site to access data stored at the NDC server terminator site;

FIG. **2** is a block diagram that provides another way of illustrating the networked, multi-processor digital computer system of FIG. 1;

FIG. **3** is a block diagram depicting a structure of the NDC included in each NDC site of FIG. 1 including the NDC's buffers;

FIG. **4**, made up of FIGS. **4A** and **4B**, is a computer program listing written in the C programming language setting forth a data structure of a channel and of a subchannel included in the channel that are used by the NDC of FIG. **3**;

FIG. **5** is a table written in the C programming language that specifies the values of various flags used by the channel illustrated in FIG. **4**;

FIG. **6** is a table written in the C programming language that defines the values of various flags used in specifying the state of channels;

FIG. **7** is a block diagram illustrating projected images of a single dataset being transferred through the NDC site depicted in FIG. **3** and illustrating the storage of various segments of the dataset in the NDC buffers;

FIG. **8** is a block diagram depicting a channel and a plurality of subchannels operating to access various segments of a dataset that have been projected into the NDC buffers illustrated in FIGS. **3** and **7**;

FIG. **9** is a table written in the C programming language defining the message type codes for the various different Data Transfer Protocol ("DTP") messages that can be transmitted between NDC sites;

FIG. **10**, made up of FIGS. **10**A and **10**B, is a definition written in the C programming language of the data structure for DTP messages;

FIG. **11**, made up of FIGS. **11**A, **11**B, **11**C, **11**D, **11**E, **11**F, **11**G, **11**H, and **11**I, are definitions written in the C programming language for various data sub-structures incorporated into the structures of FIGS. **4** and **10**;

FIG. **12** is a definition written in the C programming language of the data structure that is used in chaining together DTP messages;

FIG. **13**, made up of FIGS. **13**A and **13**B, is a definition written in the C programming language for a data structure that contains the channel's metadata;

FIG. **14** is a definition written in the C programming language setting forth the structure of an upstream site structure that is used by the NDC of FIGS. **3** and **7** for storing information about the activity of upstream NDC sites in accessing a dataset stored at the NDC server terminator site;

FIG. **15** is a block diagram illustrating a tree of NDC sites including an NDC server terminator site having a stored file that may be accessed from a plurality of NDC client terminator sites; and

FIG. **16** is a block diagram illustrating the application of the NDC within a file server employing a network of digital computers.

BEST MODE FOR CARRYING OUT THE INVENTION

FIG. **1** is a block diagram depicting a networked, multiprocessor digital computer system referred to by the general reference character **20**. The digital computer system **20** includes a Network Distributed Cache ("NDC™") server site **22**, an NDC client site **24**, and a plurality of intermediate NDC sites **26**A and **26**B. Each of the NDC sites **22**, **24**, **26**A and **26**B in the digital computer system **20** includes a processor and RAM, neither of which are illustrated in FIG. 1. Furthermore, the NDC server site **22** includes a hard disk **32** for storing data that may be accessed by the client site **24**. The NDC client site **24** and the intermediate NDC site **26**B both include their own respective hard disks **34** and **36**. A client workstation **42** communicates with the NDC client site **24** via an Ethernet Local Area Network ("LAN") **44** in accordance with a network protocol such as that of the NFS systems identified above.

Each of the NDC sites **22**, **24**, **26**A and **26**B in the networked computer system **20** includes an NDC **50**, an enlarged version of which is depicted for intermediate site **26**A. The NDCs **50** in each of the NDC sites **22**, **24**, **26**A and **26**B include a set of computer programs and a data cache located in the RAM of the NDC sites **22**, **24**, **26**A and **26**B. The NDCs **50** together with Data Transfer Protocol ("DTP™") messages **52**, illustrated in FIG. **1** by the lines joining pairs of NDCs **50**, provide a data communication network by which the client workstation **42** may access data on the hard disk **32** via the NDC sites **24**, **26**B, **26**A and **22**.

The NDCs **50** operate on a data structure called a "dataset." Datasets are named sequences of bytes of data that are addressed by:

a server-id that identifies the NDC server site where source data is located, such as NDC server site **22**; and

a dataset-id that identifies a particular item of source data stored at that site, usually on a hard disk, such as the hard disk **32** of the NDC server site **22**.

The dataset-id may specify a file on the hard disk **32** of the NDC server site **22**, in which case it would likely be a compound identifier (filesystem id, file id), or it may specify any other contiguous byte sequence that the NDC server site **22** is capable of interpreting and is willing to transmit to the NDC client site **24**. For example, a dataset could be ten pages from the RAM of the NDC server site **22**. Such a ten page segment from the RAM of the NDC server site **22** might itself be specified with a filesystem-id that identifies virtual memory and a file-id that denoted the starting page number within the virtual memory.

The NDC client site **24** requests access to data from the NDC server site **22** using an NDC_LOAD message specifying whether the type of activity being performed on the dataset at the NDC client site **24** is a read or a write operation. The range of data requested with an NDC_LOAD message specifies the byte sequences within the named dataset that are being accessed by the NDC client site **24**. A single request by the NDC client site **24** may specify several disparate byte sequences, with no restriction on the size of each sequence other than it be discontiguous from all other sequences specified in the same request. Thus, each request to access data by the NDC client site **24** contains a series of range specifications, each one of which is a list of offset/length pairs that identify individual contiguous byte sequences within the named dataset.

Topology of an NDC Network

An NDC network,such as that illustrated in FIG. 1 having NDC sites **22**, **24**, **26**A and **26**B, includes:

1. all nodes in a network of processors that are configured to participate as NDC sites; and
2. the DTP messages **52** that bind together NDC sites, such as NDC sites **22**, **24**, **26**A and **26**B.

Any node in a network of processors that possesses a megabyte or more of surplus RAM may be configured as an NDC site. NDC sites communicate with each other via the DTP messages **52** in a manner that is completely compatible with non-NDC sites.

FIG. 1 depicts a series of NDC sites **22**, **24**, **26**A and **26**B linked together by the DTP messages **52** that form a chain connecting the client workstation **42** to the NDC server site **22**. The NDC chain may be analogized to an electrical transmission line. The transmission line of the NDC chain is terminated at both ends, i.e., by the NDC server site **22** and by the NDC client site **24**. Thus, the NDC server site **22** may be referred to as an NDC server terminator site for the NDC chain, and the NDC client site **24** may be referred to as an NDC client terminator site for the NDC chain. An NDC server terminator site **22** will always be the node in the network of processors that "owns" the source data structure. The other end of the NDC chain, the NDC client terminator site **24**, is the NDC site that receives requests from the client workstation **42** to access data on the NDC server site **22**.

Data being written to the hard disk **32** at the NDC server site **22** by the client workstation **42** flows in a "downstream" direction indicated by a downstream arrow **54**. Data being loaded by the client workstation **42** from the hard disk **32** at the NDC server site **22** is pumped "upstream" through the NDC chain in the direction indicated by an upstream arrow

56 until it reaches the NDC client site **24**. When data reaches the NDC client site **24**, it together with metadata is reformatted into a reply message in accordance with the appropriate network protocol such as NFS, and sent back to the client workstation **42**. NDC sites are frequently referred to as being either upstream or downstream of another NDC site. The downstream NDC site **22**, **26A** or **26B** must be aware of the types of activities being performed at its upstream NDC sites **26A**, **26B** or **24** at all times.

In the network depicted in FIG. **1**, a single request by the client workstation **42** to read data stored on the hard disk **32** is serviced in the following manner:

1. The request flows across the Ethernet LAN **44** to the NDC client site **24** which serves as a gateway to the NDC chain. Within the NDC client site **24**, an NDC client intercept routine **102**, illustrated in FIGS. **3** and **7**, inspects the request. If the request is an NFS request and if the request is directed at any NDC site **24**, **26A**, **26B**, or **22** for which the NDC client site **24** is a gateway, then the request is intercepted by the NDC client intercept routine **102**.
2. The NDC client intercept routine **102** converts the NFS request into a DTP request, and then submits the request to an NDC core **106**.
3. The NDC core **106** in the NDC client site **24** receives the request and checks its NDC cache to determine if the requested data is already present there. If all data is present in the NDC cache of the NDC client site **24**, the NDC **50** will copy pointers to the data into a reply message structure and immediately respond to the calling NDC client intercept routine **102**.
4. If all the requested data isn't present in the NDC cache of the NDC client site **24**, then the NDC **50** will access any missing data elsewhere. If the NDC site **24** were a server terminator site, then the NDC **50** would access the filesystem for the hard disk **34** upon which the data would reside.
5. Since the NDC client site **24** is a client terminator site rather than a server terminator site, the NDC **50** must request the data it needs from the next downstream NDC site, i.e., intermediate NDC site **26B** in the example depicted in FIG. **1**. Under this circumstance, DTP client interface routines **108**, illustrated in FIGS. **3** and **7**, are invoked to request from the intermediate NDC site **26B** whatever additional data the NDC client site **24** needs to respond to the current request.
6. A DTP server interface routine **104**, illustrated in FIGS. **3** and **7**, at the downstream intermediate NDC site **26B** receives the request from the NDC **50** of the NDC client site **24** and processes it according to steps **3**, **4**, and **5** above. The preceding sequence repeats for each of the NDC sites **24**, **26B**, **26A** and **22** in the NDC chain until the request reaches the server terminator, i.e., NDC server site **22** in the example depicted in FIG. **1**, or until the request reaches an NDC site that has all the data that is being requested of it.
7. When the NDC server terminator site **22** receives the request, its NDC **50** accesses the source data structure. If the source data structure resides on a hard disk, the appropriate file system code (UFS, DOS, etc.) is invoked to retrieve the data from the hard disk **32**.
8. When the file system code on the NDC server site **22** returns the data from the hard disk **32**, a response chain begins whereby each downstream site successively responds upstream to its client, e.g. NDC server site **22** responds to the request from intermediate NDC site **26A**, intermediate NDC site **26A** responds to the request from intermediate NDC site **26B**, etc.
9. Eventually, the response percolates up through the sites **22**, **26A**, and **26B** to the NDC client terminator site **24**.
10. The NDC **50** on the NDC client site **24** returns to the calling NDC client intercept routine **102**, which then packages the returned data and metadata into an appropriate network protocol format, such as that for an NFS reply, and sends the data and metadata back to the client workstation **42**.

The NDC client intercept routines **102** are responsible for performing all conversions required between any supported native protocol, e.g. NFS, Server Message Block ("SMB"), Novelle Netware®, etc., and the DTP messages **52** employed for communicating among the NDCs **50** making up the NDC chain. The conversion between each native protocol and the DTP messages **52** must be so thorough that client workstations, such as the client workstation **42**, are unable to distinguish any difference in operation between an NDC **50** functioning as a server to that workstation and that workstation's "native" server.

An alternative way of visualizing the operation of the NDCs **50'** is illustrated in FIG. **2**. Those elements depicted in FIG. **2** that are common to the digital computer system **20** depicted in FIG. **1** bear the same reference numeral distinguished by a prime ("'") designation. The NDCs **50'** in the sites **22'**, **26A'**, **26B'** and **24'** provide a very high speed data conduit **62** connecting the client intercept routines **102** of the NDC client site **24'** to file system interface routines **112** of the NDC server site **22'**, illustrated in FIGS. **3** and **7**. Client workstations, using their own native protocols, may plug into the data conduit **62** at each of the NDC sites **22**, **26A**, **26B** and **24** via the NDC's client intercept routines **102** in each of the NDC sites **22**, **26A**, **26B** and **24**. Accordingly, the NDC **50** of the intermediate NDC site **26A** may interface into a Novelle Netware network **64**. Similarly, the NDC **50** of the intermediate NDC site **26B** may interface into a SMB network **66**, and into an NFS network **68**. If an NDC site **24**, **26B**, **26A** or **22** is both the client terminator site and the server terminator site for a request to access data, then the NDC data conduit **62** is contained entirely within that NDC site **24**, **26B**, **26A** or **22**.

After an NDC **50'** intercepts a request from a client workstation on one of the networks **44'**, **64**, **66** or **68** and converts it into the DTP messages **52'**, the request travels through the data conduit **62** until all the data has been located. If a request is a client's first for a particular dataset, the DTP messages **52'** interconnecting each pair of NDCs **50'** form the data conduit **62** just in advance of a request's passage. If a request reaches the NDC server terminator site **22'**, the NDC **50'** directs it to the appropriate file system on the NDC server terminator site **22'**. Each NDC site **22'** may support several different types of file systems for hard disks attached thereto such as the hard disks **32'**, **34'**, and **36'**.

After the file system at the NDC server terminator site **22'** returns the requested data to its NDC **50'**, the NDC **50'** passes the reply data and metadata back up through each NDC site **26A'** and **26B'** until it reaches the client terminator **24'**. At the client terminator **24'**, the NDC routine originally called by the NDC client intercept routine **102** returns back to that routine. The NDC client intercept routine **102** then reformats the data and metadata into an appropriately formatted reply message and dispatches that message back to the client workstation **42'**.

Four components of the NDC **50'** support the data conduit **62**:

The resource management mechanisms of the NDC client terminator site that measure the rate at which its client